Understanding Linux NUMA Memory Policy

Lee Schermerhorn
HP/OSLO Scalability and Performance Team

Agenda

- NUMA and Memory Locality
- NUMA Memory Policy in Linux
  - Hierarchy of Memory Policies
  - Components of Memory Policies
  - Policy for Page Cache Pages
  - Node lists and node overflow
- libnuma API
- numactl CLI
- cpusets
- /proc/<pid>/numa_maps

NUMA Systems and Memory Locality

- Why build systems with Non-Uniform Memory Access times?
  - to provide sufficient memory/IO/interconnect bandwidth to achieve scaling with modern, high-performance processors
- How?
  - multiple low cpu count SMP/SMT building blocks ["nodes" or "cells"] with sufficient intra-node bandwidth for local cpus, IO subsystem and the system interconnect.
  - low latency inter-node interconnect of sufficient bandwidth to support off-node traffic
- Heavily dependent on locality for a "win", similar to processor caches

How Do We Achieve NUMA Locality?

- Enhance the kernel to "do the right thing"
  - reasonable [non-pathological] default behaviour
    - for user programs' execution cpu and memory location.
    - for algorithms and kernel data structure design and placement
  - Tunable behaviour for various application workloads
- Mechanisms to provide hints or application specific information to kernel:
  - Inheritable or "other-directed" behaviour/options settable by command line tools—e.g., numactl—for unmodified applications
  - APIs—e.g., libnuma—for NUMA aware applications and language run-time environments. Requires source modification or, at least, recompile/relink.

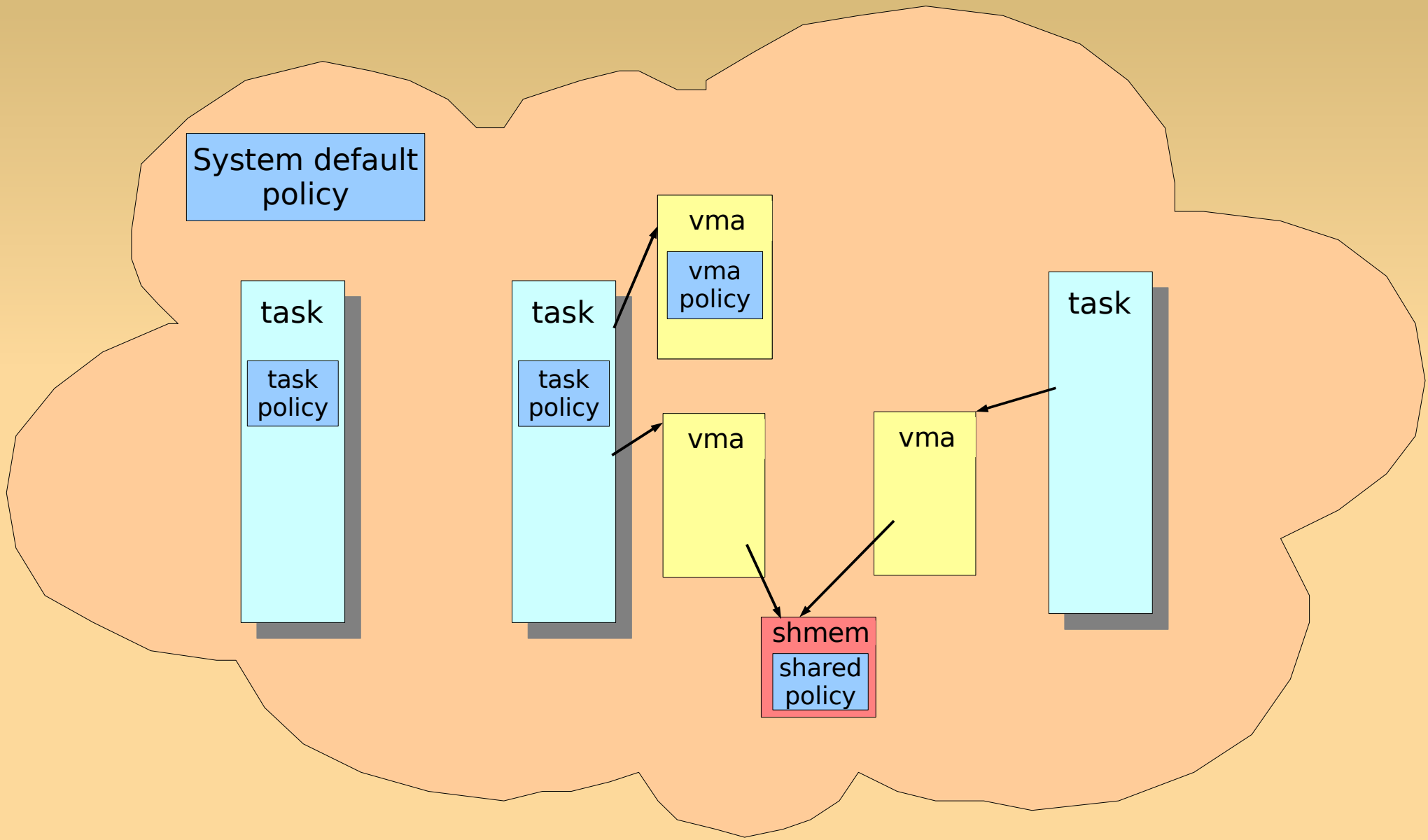
NUMA Memory Policy in Linux

- Since ~2.6.6, Linux has implemented control of memory allocation on NUMA platforms using "memory policies"
  - Original kernel patches and libnuma by Andi Kleen of SuSE
  - available in:
    - 2.6.5-based SLES9
    - 2.6.9-based RHEL4
  - Enhancements and additional capabilities since then, but basic mechanism and APIs have remained the same.
- Memory Policy determines on which node the kernel will allocate a page or kernel data structure

Hierarchy of Memory Policies

- System-wide default policy ⇨ allocate from node local to cpu where allocation occurs
- Task [process] policy
  - applies to all allocations in task that aren't governed by another policy, below
  - inherited by children on fork() and across exec*()
- Per "vm area" [region of virtual address space] policy
  - applies only to anonymous memory regions, and "COWed" pages of private file mappings.
  - also inherited on fork(). NOT across exec*().
- Shared Policies:
  - shared memory segments, mapped tmpfs and hugetlbfs files

The Policy Hierarchy...



Policy for Page Cache Pages

- Page Cache contains:
  - file pages read or written via [buffered] `read(2)`, `write(2)` and related system calls;
  - pages of regular files mapped via `mmap(2)` system call.
- Page cache pages are always allocated using task policy of task which causes allocation
  - remember, falls back to system default == local, if no explicit task policy
- For private mappings, an anonymous page will be allocated that follows "vma policy", if any:
  - if/when the page is written by task
  - original file page remains in page cache, where allocated

Components of Memory Policies

- "mode" [a.k.a. policy] == "behavior"
  - `MPOL_DEFAULT` – context dependent meaning
    - system default ⇨ local allocation
    - task default ⇨ use system default == local
    - vma default ⇨ use task default – maybe != local
  - `MPOL_BIND` – allocate only from specified list of nodes
  - `MPOL_PREFERRED` – attempt allocation from specified node first
  - `MPOL_INTERLEAVE` – interleave allocations [page granularity] across specified nodes; node id order
    - vma policy: based on address offset into region
    - task policy: based on free running allocation counter
- Optional set of nodes
  - default: no node[s]; preferred: one node; bind, interleaved: one or more nodes

Node Lists and Node Overflow

- Kernel constructs, for each node, a node list for page allocation "overflow":
  - starting with the node itself
  - remaining nodes, ordered by "distance" as reported by the ACPI "SLIT":
    - "System Locality Information Table"
    - created by platform firmware
- Lists are used when no memory is available on a node to satisfy page allocation request
  - traverse list to find first node that satisfies request
- Custom lists created for `MPOL_BIND` policies:
  - order is unspecified
    - implementation: node id numeric order

libnuma APIs - set_mempolicy()

```
#include <numaif.h>
long set_mempolicy(int mode, unsigned long *nodemask,
                   unsigned long maxnode);
```

- Sets task/process policy to specified mode with node[s] from nodemask, where required.
  - nodemask may be an array containing at least maxnode bits.
    - should be NULL for MPOL_DEFAULT
    - For MPOL_PREFERRED:
      - first [lowest numeric] node used, if more than 1 specified
      - local node used if nodemask == NULL
- Policy applied when a page is allocated
- Not remembered if page is swapped/paged out
  - and, e.g., task policy has changed since allocation

libnuma APIs - get_mempolicy() - 1

```
#include <numaif.h>
long get_mempolicy(int *mode, unsigned long *nodemask,
                   unsigned long maxnode, void *addr, int flags);
```

- Query NUMA policy of task/process or of a specified memory address
  - policy mode is stored in mode, if defined, and nodes are stored in nodemask.
  - maxnode is size of nodemask in bits
- Flags:
  - MPOL_F_NODE – store [in mode] next interleave node if policy is MPOL_INTERLEAVE, or when used with...
  - MPOL_F_ADDR – fetch node location of addr.

libnuma APIs - get_mempolicy() - 2

```
#include <numaif.h>

/*
 * get_node() -- fetch numa node id of page at vaddr
 * If no page allocated at vaddr, will cause page to fault it in according
 * to policy.  If anon page, fault will behave like a "read" access and
 * install the shared "ZEROPAGE" in the kernel image.
 */
static int
get_node(void *vaddr)
{
        int rc, node;

        rc = get_mempolicy(&node, NULL, 0, vaddr, MPOL_F_NODE|MPOL_F_ADDR);
        if (rc)
                return -1;

        return node;
}
```

libnuma APIs - mbind() - 1

```
#include <numaif.h>
long mbind(void *start, unsigned long len, int *mode,
          unsigned long *nodemask, unsigned long maxnode,
          unsigned flags);
```

- set NUMA policy for specified range to mode + nodemask
  - sets "vma policy" for that range
  - splits virtual memory area if ranges specifies a subset of a previously mapped area
    - Shared memory regions, tmpfs and hugetlbfs mappings support multiple, shared policies on subsets of region/mapping.
- policy applies only to pages allocated after the mbind() call.
    - unless migration supported and requested...

libnuma APIs - mbind() - 2

- mbind() Flags:
  - RHEL4/SLES9 [pre-2.6.16]: `MPOL_MF_STRICT` – verify that no pages already allocated in range that violate the policy.
    - Can't migrate pages in these releases.
    - return error if any pages violate policy
  - RHEL5/SLES10: added `MPOL_MF_MOVE` – move any existing pages, if necessary, to match policy.
    - restricted to pages that are referenced only by the calling tasks page tables.
    - **`MPOL_MF_MOVE_ALL`** – privileged version to migrate pages regardless of number of page table references. I.e., even if page exists in multiple tasks page tables.

libnuma APIs - other

- Two other page migration APIs:
  - migrate_pages() - migrate a specified task from one set of nodes to another
  - move_pages() - migrate specific pages in a specified task.
- These APIs not strictly related to "memory policies"

libnuma CLI - numactl - 1

```
numactl [–interleave=nodes] [–preferred=node] [–membind=nodes]
        [–localalloc] [–cpubind=nodes] <command> <args>
```

- Set task policy for <command> and any of its descendants
  - 'nodes' arguments may be specified as lists or ranges: `0,1,2,5,6,7` or `0-2,5-7`
  - sets task policy of of numactl via `set_mempolicy()`, then `fork()`s/`exec()`s <command>
  - `cpubind sets task cpu affinity for <command> via sched_setaffinity()`
    - <command> may run on and load balance between any and all cpus in the specified node[s].
  - Unless characterizing remote latency/penalty, you probably want `cpubind` nodes to be the same as, or a subset of, `membind` nodes.

libnuma CLI - numactl - 2

```
numactl –show
numactl –hardware
```

- show: display policy that numactl inherited from its parent
  - what policy has been imposed on your shell?
  - what will another `numactl` command do? E.g.
    - `numactl –membind=X –cpubind=Y numactl –show`
- hardware: show total/free memory on nodes of system.
  - and SLIT table...

libnuma CLI - numactl - 3

```
numactl [–huge] [–offset <offset>] [–mode <shmmode>]
        [–length <length>[kmg]] [–strict]
        {–shmid <id> | –shm <shmkeyfile> | –file <tmpfsfile>}
        [–touch] [–dump] <memory policy>
```

- Set policy for SysV shared memory segment or tmpfs/hugetlbfs file.
  - huge applies only to shmid/shm
  - mode, length: shmem segment attributes
  - strict: give error if pages in pre-existing region don't obey policy
  - offset: where, in the segment, the policy applies
  - touch: touch region to allocate pages
  - dump: dump [show] policy in region
  - <memory policy>: same as for launching commands
    - --interleaved, --preferred, etc.

libnuma CLI – numactl – 4 – A Quiz!

```
numactl [–membind=4-7] [–cpubind=4-7] <command> <args>
```

- What does this do?
  - Run <command> on cpus from nodes 4-7, with local allocation constrained to same 4 nodes?
- NO! It means:
  - Run <command> on cpus from nodes 4-7; allocate memory from node 4. When that overflows, move on to node 5, etc.
  - Remember, BIND policy allocates custom node list.
  - AND tasks are free to load balance between all cpus on those nodes.
  - Maybe NOT what you want/expect?
  - Consider cpusets...

cpusets - 1

- Feature in SLES9/10, RHEL5 and upstream:
  - resource partitioning and behaviour encapsulation
  - partition cpus and [nodes'] memories into cpusets
  - enable/disable specific behaviours in cpusets
  - tasks assigned to cpusets are restricted to cpus and memories regardless of policies.
    - policies "masked"/constrained by cpuset restrictions
  - hierarchy of cpusets: children subdivide parent's resources or specify different behaviours with same resources;
    - can overlap unless set '{cpu|mem}_exclusive'
  - user interface is cpuset file system:
    - mount -t cpuset none /cpusets
    - mount point == root cpuset. contains all cpus/mems

cpusets - 2

- More user interface
  - mkdir makes child cpuset. e.g.,
    - mkdir /cpusets/nodes2-3
  - write values to 'cpus' and 'mems' pseudo-files to assign resources:
    - echo 8-15 >/cpusets/nodes2-3/cpus
    - echo 2,3 >/cpusets/nodes2-3/mems
  - write pids to 'tasks' pseudo-file to assign tasks to cpusets. E.g., to assign my shell to nodes2-3:
    - echo $$ >/cpusets/nodes2-3/tasks
  - read any of these pseudo-files to examine:
    - cat /cpusets/nodes2-3/tasks
  - read /proc/<pid>/cpuset to query a task's cpuset:
    - cat /proc/$$/cpuset

cpusets - 3

- Behaviour encapsulation vis à vis memory policy:
  - memory_spread_page: spread/interleave page cache across mems in cpuset, instead of using task policy
    - in SLES9: ??? [different name]
  - memory_spread_slab: spread kernel allocations across mems, instead of using task policy
  - memory_migrate: migrate task memory when added to cpuset or when cpuset's mems change

/proc/<pid>/numa_maps

```
#cat /proc/$$/numa_maps
<snip>
2000000000088000 default file=/lib/tls/libc-2.3.4.so mapped=77 mapmax=38 N4=77
2000000002dc000 default file=/lib/tls/libc-2.3.4.so
2000000002e8000 default file=/lib/tls/libc-2.3.4.so anon=2 dirty=2 N0=1 N1=1
2000000002f0000 default anon=7 dirty=7 N0=1 N1=6
<snip>
```

- Examine NUMA policy and page allocation for task <pid> address space
  - virtual address, policy [default], mapped file
  - anon/dirty page count
  - number of mapped pages, max mapcount [pte references]
  - NX=<pagecount> shows number of pages on node X for this virtual memory area

Any [more] Questions?

Understanding Linux NUMA Memory Policy

Lee Schermerhorn
HP/OSLO Scalability and Performance Team

Agenda

- NUMA and Memory Locality
- NUMA Memory Policy in Linux
  - Hierarchy of Memory Policies
  - Components of Memory Policies
  - Policy for Page Cache Pages
  - Node lists and node overflow
- libnuma API
- numactl CLI
- cpusets
- /proc/<pid>/numa_maps

18/04/07 Linux Memory Policy V0.1 2

Numa and Memory locality => motivation – why built numa platforms and why locality matters.
Memory policy – concepts

libnuma – APIs for tools and applications to manipulate memory policy

numactl – CLI built on libnuma – external control of unmodified apps' locality

cpusets – trump policy – administrative constraints on resources available to policy control

numa_maps => seeing what's happening

NUMA Systems and Memory Locality

- Why build systems with Non-Uniform Memory Access times?
  - to provide sufficient memory/IO/interconnect bandwidth to achieve scaling with modern, high-performance processors
- How?
  - multiple low cpu count SMP/SMT building blocks ["nodes" or "cells"] with sufficient intra-node bandwidth for local cpus, IO subsystem and the system interconnect.
  - low latency inter-node interconnect of sufficient bandwidth to support off-node traffic
- Heavily dependent on locality for a "win", similar to processor caches

18/04/07 Linux Memory Policy V0.1 3

Why build NUMA platforms in the first place?

Technical difficulty [=> expense] of scalable system interconnects

Take away: Locality Matters...

How Do We Achieve NUMA Locality?

- Enhance the kernel to "do the right thing"
  - reasonable [non-pathological] default behaviour
    - for user programs' execution cpu and memory location.
    - for algorithms and kernel data structure design and placement
  - Tunable behaviour for various application workloads
- Mechanisms to provide hints or application specific information to kernel:
  - Inheritable or "other-directed" behaviour/options settable by command line tools—e.g., numactl—for unmodified applications
  - APIs—e.g., libnuma—for NUMA aware applications and language run-time environments. Requires source modification or, at least, recompile/relink.

18/04/07 Linux Memory Policy V0.1 4

NUMA platforms require OS awareness to avoid worst case behaviour on NUMA platforms.

One size does NOTfit all applications – need tunable behavior of OS, and/or APIs/CLIs to specific application specific behaviour and memory placement.

NUMA Memory Policy in Linux

- Since ~2.6.6, Linux has implemented control of memory allocation on NUMA platforms using "memory policies"
  - Original kernel patches and libnuma by Andi Kleen of SuSE
  - available in:
    - 2.6.5-based SLES9
    - 2.6.9-based RHEL4
  - Enhancements and additional capabilities since then, but basic mechanism and APIs have remained the same.
- Memory Policy determines on which node the kernel will allocate a page or kernel data structure

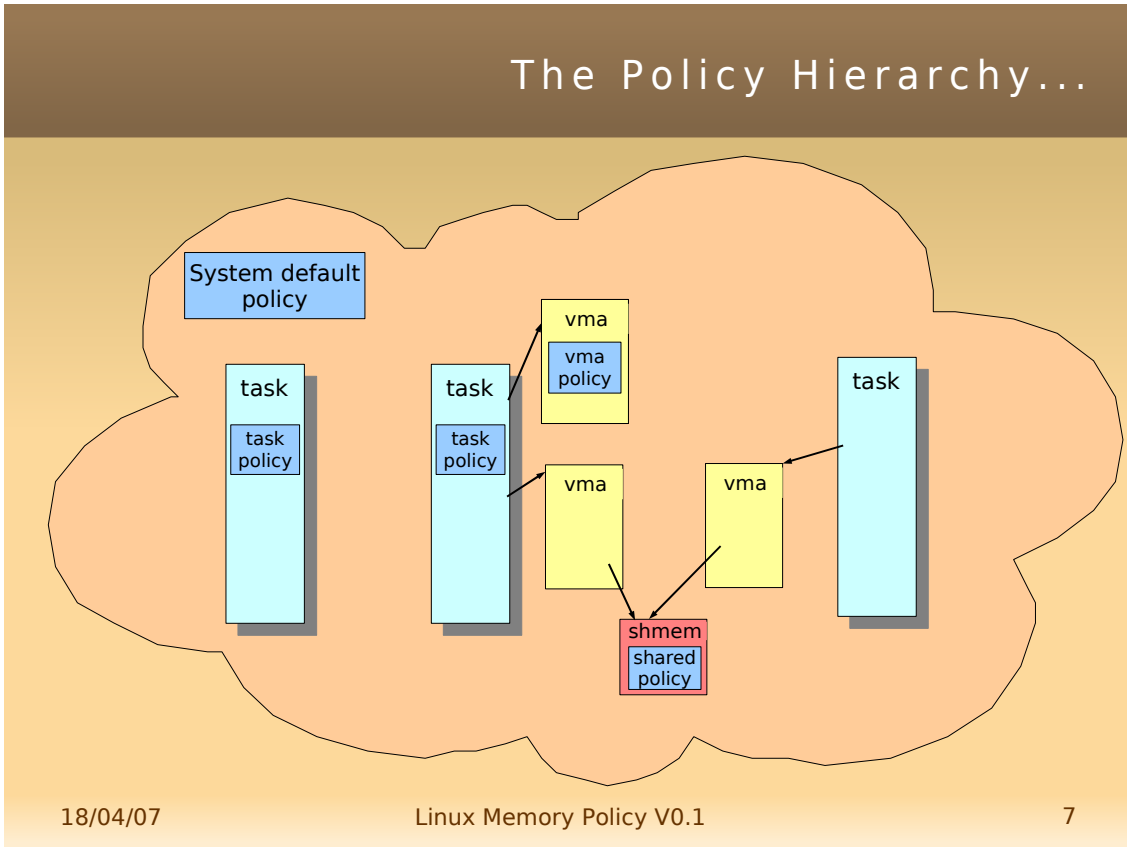
Linux Memory Policy concepts

Hierarchy of Memory Policies

- System-wide default policy ⇨ allocate from node local to cpu where allocation occurs
- Task [process] policy
  - applies to all allocations in task that aren't governed by another policy, below
  - inherited by children on fork() and across exec*()
- Per "vm area" [region of virtual address space] policy
  - applies only to anonymous memory regions, and "COWed" pages of private file mappings.
  - also inherited on fork(). NOT across exec*().
- Shared Policies:
  - shared memory segments, mapped tmpfs and hugetlbfs files

18/04/07 Linux Memory Policy V0.1 6

* actually a hierarchry of "policy scope"
* If no task/process policy specified, falls back to system default policy.
* If no "vma policy" specified for a virtual memory address, falls back to task policy, if any. Also falls back to task policy when vma policy == "default". More later
* COW == Copy-on-Write
* Note that vma policy does not apply to shared, mapped file pages: includes all pages in shared mappings and pages in private mappings that have not been written by task. More later.
* Shared policies: shared between all mappers of "object". support different policies on different ranges of "object"--e.g., shared memory region; mapped tmpfs file, ... NOT supported for shared mappings of regular files :-(



An attempt to visualize the various policy scopes

Policy for Page Cache Pages

- Page Cache contains:
  - file pages read or written via [buffered] `read(2)`, `write(2)` and related system calls;
  - pages of regular files mapped via `mmap(2)` system call.
- Page cache pages are always allocated using task policy of task which causes allocation
  - remember, falls back to system default == local, if no explicit task policy
- For private mappings, an anonymous page will be allocated that follows "vma policy", if any:
  - if/when the page is written by task
  - original file page remains in page cache, where allocated

* Only control over page cache pages is by setting task's mempolicy before reading/writing file. And then, only has effect if pages not already in the cache.

Components of Memory Policies

- "mode" [a.k.a. policy] == "behavior"
  - MPOL_DEFAULT – context dependent meaning
    - system default ⇨ local allocation
    - task default ⇨ use system default == local
    - vma default ⇨ use task default – maybe != local
  - MPOL_BIND – allocate only from specified list of nodes
  - MPOL_PREFERRED – attempt allocation from specified node first
  - MPOL_INTERLEAVE – interleave allocations [page granularity] across specified nodes; node id order
    - vma policy: based on address offset into region
    - task policy: based on free running allocation counter
- Optional set of nodes
  - default: no node[s]; preferred: one node; bind, interleaved: one or more nodes

18/04/07 Linux Memory Policy V0.1 9

* mode vs policy: man pages call it policy. function prototypes in /usr/include/numaif.h vary:
  "policy" in get_mempolicy() prototype
  "mode" in set_mempolicy() and mbind() prototype
  Of course, names in prototypes are optional and ignored anyway.
* Bind, Preferred: more info follows on how multiple nodes used for Bind and how Bind and Preferred overflow
* Interleaved: interleaves pages to nodes in node id numeric order [implementation is bit mask]

Node Lists and Node Overflow

- Kernel constructs, for each node, a node list for page allocation "overflow":
  - starting with the node itself
  - remaining nodes, ordered by "distance" as reported by the ACPI "SLIT":
    - "System Locality Information Table"
    - created by platform firmware
- Lists are used when no memory is available on a node to satisfy page allocation request
  - traverse list to find first node that satisfies request
- Custom lists created for `MPOL_BIND` policies:
  - order is unspecified
    - implementation: node id numeric order

* Some "internals" info: important for understanding how allocation works when nodes memory fills up.
* Preferred policy starts with a specified node and overflows along node list constructed at boot time [node hot add time?]
* Interleave: similarly—if no memory in selected node, overflow along node's node list
* Bind: use custom node list. Always start at lowest numbered node, and overflow along custom list in node id order.
* Overflow: really a bit more complicated: may attempt to reclaim pages—e.g., from page cache, or by swapping out inactive anon pages—before overflowing to another node.

libnuma APIs - set_mempolicy()

```
#include <numaif.h>
long set_mempolicy(int mode, unsigned long *nodemask,
                   unsigned long maxnode);
```

- Sets task/process policy to specified mode with node[s] from nodemask, where required.
  - nodemask may be an array containing at least maxnode bits.
    - should be NULL for MPOL_DEFAULT
    - For MPOL_PREFERRED:
      - first [lowest numeric] node used, if more than 1 specified
      - local node used if nodemask == NULL
- Policy applied when a page is allocated
- Not remembered if page is swapped/paged out
  - and, e.g., task policy has changed since allocation

18/04/07 Linux Memory Policy V0.1 11

* Man page may be wrong w/rt to return type. See the definition in the header.

* Can use task policy as a "cursor" for allocating memory on different nodes. E.g., set policy to bind to a specific node and touch/allocate some pages, set to a different node and allocate some more.
* BUT, policy won't be remember across page out/in, so need to lock into memory to prevent this, if important. Or use mbind() for vma policy – more later – but this has limitations as well.

* Note: for anonymous memory, must write [store] to allocate. Initial reads/loads will only map the common "zero page". Allocation only occurs on first store/write.

libnuma APIs - get_mempolicy() - 1

```
#include <numaif.h>
long get_mempolicy(int *mode, unsigned long *nodemask,
                   unsigned long maxnode, void *addr, int flags);
```

- Query NUMA policy of task/process or of a specified memory address
  - policy mode is stored in `mode`, if defined, and nodes are stored in `nodemask`.
  - `maxnode` is size of `nodemask` in bits
- Flags:
  - `MPOL_F_NODE` – store [in `mode`] next interleave node if policy is `MPOL_INTERLEAVE`, or when used with...
  - `MPOL_F_ADDR` – fetch node location of `addr`.

* Man page may be wrong w/rt to return type. See the definition in the header.

* Use MPOL_F_ADDR|MPOL_F_NODE to query node where a page is allocated. Note: page will be faulted in/allocated, if necessary, to satisfy this request. This fault behaves like a read fault.

libnuma APIs - get_mempolicy() - 2

```
#include <numaif.h>

/*
 * get_node() -- fetch numa node id of page at vaddr
 * If no page allocated at vaddr, will cause page to fault it in according
 * to policy.  If anon page, fault will behave like a "read" access and
 * install the shared "ZEROPAGE" in the kernel image.
 */
static int
get_node(void *vaddr)
{
        int rc, node;

        rc = get_mempolicy(&node, NULL, 0, vaddr, MPOL_F_NODE|MPOL_F_ADDR);
        if (rc)
                return -1;

        return node;
}
```


A useful [?] wrapper around get_mempolicy() to fetch node location of page at a virtual address.

libnuma APIs - mbind() - 1

```
#include <numaif.h>
long mbind(void *start, unsigned long len, int *mode,
          unsigned long *nodemask, unsigned long maxnode,
          unsigned flags);
```

- set NUMA policy for specified range to mode + nodemask
  - sets "vma policy" for that range
  - splits virtual memory area if ranges specifies a subset of a previously mapped area
    - Shared memory regions, tmpfs and hugetlbfs mappings support multiple, shared policies on subsets of region/mapping.
- policy applies only to pages allocated after the mbind() call.
    - unless migration supported and requested...

* Note: 'vma' is name of internal structure that tracks ranges of virtual address space.
* cat /proc/<pid>/maps to see list of vmas.
* mbind() of range of va that is subset of existing vma will split vma to install policy on that range.

libnuma APIs - mbind() - 2

- mbind() Flags:
  - RHEL4/SLES9 [pre-2.6.16]: `MPOL_MF_STRICT` – verify that no pages already allocated in range that violate the policy.
    - Can't migrate pages in these releases.
    - return error if any pages violate policy
  - RHEL5/SLES10: added `MPOL_MF_MOVE` – move any existing pages, if necessary, to match policy.
    - restricted to pages that are referenced only by the calling tasks page tables.
    - **`MPOL_MF_MOVE_ALL`** – privileged version to migrate pages regardless of number of page table references. I.e., even if page exists in multiple tasks page tables.

* re: page table references: use this term to avoid confusion with term "mapped". A region may be mapped into a task's address space—e.g., via mmap()—without any pages being allocated, or with the pages allocated, perhaps by another task, but with no page table entries loaded for "this" task. If multiple tasks map a region and touch the pages, then page table entries will be created for each task. The kernel counts these entries/references using a struct page member called, unfortunately, 'mapcount'. Don't want to confuse this with the mapping or unmapping of regions via m[un]map().

libnuma APIs - other

- Two other page migration APIs:
  - migrate_pages() - migrate a specified task from one set of nodes to another
  - move_pages() - migrate specific pages in a specified task.
- These APIs not strictly related to "memory policies"

libnuma CLI - numactl - 1

```
numactl [–interleave=nodes] [–preferred=node] [–membind=nodes]
        [–localalloc] [–cpubind=nodes] <command> <args>
```

- Set task policy for <command> and any of its descendants
  - 'nodes' arguments may be specified as lists or ranges: `0,1,2,5,6,7` or `0-2,5-7`
  - sets task policy of of numactl via `set_mempolicy()`, then `fork()`s/`exec()`s <command>
  - `cpubind sets task cpu affinity for <command> via sched_setaffinity()`
    - <command> may run on and load balance between any and all cpus in the specified node[s].
  - Unless characterizing remote latency/penalty, you probably want `cpubind` nodes to be the same as, or a subset of, `membind` nodes.

18/04/07 Linux Memory Policy V0.1 17

* Command Line Interface to set the task/process policy for an unmodified application.
* As if application called set_mempolicy() first thing. Actually, if task has issued policy, any pages allocated during runtime start up would be placed based on parent policy/location. Using numactl to launch application ensures that even these early allocations come from desired node.

libnuma CLI - numactl - 2

```
numactl –show
numactl –hardware
```

- show: display policy that numactl inherited from its parent
  - what policy has been imposed on your shell?
  - what will another `numactl` command do? E.g.
    - `numactl –membind=X –cpubind=Y numactl –show`
- hardware: show total/free memory on nodes of system.
  - and SLIT table...

* '--show' not all that useful except as listed above.

libnuma CLI - numactl - 3

```
numactl [–huge] [–offset <offset>] [–mode <shmmode>]
        [–length <length>[kmg]] [–strict]
        {–shmid <id> | –shm <shmkeyfile> | –file <tmpfsfile>}
        [–touch] [–dump] <memory policy>
```

- Set policy for SysV shared memory segment or tmpfs/hugetlbfs file.
  - huge applies only to shmid/shm
  - mode, length: shmem segment attributes
  - strict: give error if pages in pre-existing region don't obey policy
  - offset: where, in the segment, the policy applies
  - touch: touch region to allocate pages
  - dump: dump [show] policy in region
  - <memory policy>: same as for launching commands
    - --interleaved, --preferred, etc.

* preallocation and placement of shared memory segments, ...

libnuma CLI – numactl – 4 – A Quiz!

```
numactl [–membind=4-7] [–cpubind=4-7] <command> <args>
```

- What does this do?
  - Run <command> on cpus from nodes 4-7, with local allocation constrained to same 4 nodes?
- NO! It means:
  - Run <command> on cpus from nodes 4-7; allocate memory from node 4. When that overflows, move on to node 5, etc.
  - Remember, BIND policy allocates custom node list.
  - AND tasks are free to load balance between all cpus on those nodes.
  - Maybe NOT what you want/expect?
  - Consider cpusets...

cpusets - 1

- Feature in SLES9/10, RHEL5 and upstream:
  - resource partitioning and behaviour encapsulation
  - partition cpus and [nodes'] memories into cpusets
  - enable/disable specific behaviours in cpusets
  - tasks assigned to cpusets are restricted to cpus and memories regardless of policies.
    - policies "masked"/constrained by cpuset restrictions
  - hierarchy of cpusets: children subdivide parent's resources or specify different behaviours with same resources;
    - can overlap unless set '{cpu|mem}_exclusive'
  - user interface is cpuset file system:
    - mount -t cpuset none /cpusets
    - mount point == root cpuset. contains all cpus/mems

* Note that children always overlap parents' resources. Siblings may only overlap if !*_exclusive

* can be used to partition cpus and launch applications similar to HP-UX processor sets.
* Suggestion: try writing shell scripts to simulate HP-UX processor set control and application launch commands.

cpusets - 2

- More user interface
  - mkdir makes child cpuset. e.g.,
    - mkdir /cpusets/nodes2-3
  - write values to 'cpus' and 'mems' pseudo-files to assign resources:
    - echo 8-15 >/cpusets/nodes2-3/cpus
    - echo 2,3 >/cpusets/nodes2-3/mems
  - write pids to 'tasks' pseudo-file to assign tasks to cpusets. E.g., to assign my shell to nodes2-3:
    - echo $$ >/cpusets/nodes2-3/tasks
  - read any of these pseudo-files to examine:
    - cat /cpusets/nodes2-3/tasks
  - read /proc/<pid>/cpuset to query a task's cpuset:
    - cat /proc/$$/cpuset

* subdirectories partition parent resources
* read control files via cat(1) or application read(2)
* write contro files via echo(1) or application write(2)

cpusets - 3

- Behaviour encapsulation vis à vis memory policy:
  - memory_spread_page: spread/interleave page cache across mems in cpuset, instead of using task policy
    - in SLES9: ??? [different name]
  - memory_spread_slab: spread kernel allocations across mems, instead of using task policy
  - memory_migrate: migrate task memory when added to cpuset or when cpuset's mems change

* AsideL upstream kernel work: generalize cpusets into task groups for "containers"--more general resource partitioning, including various namespaces [pid, IPC, sockets, file system, ...]

/proc/<pid>/numa_maps

```
#cat /proc/$$/numa_maps
<snip>
2000000000088000 default file=/lib/tls/libc-2.3.4.so mapped=77 mapmax=38 N4=77
20000000002dc000 default file=/lib/tls/libc-2.3.4.so
20000000002e8000 default file=/lib/tls/libc-2.3.4.so anon=2 dirty=2 N0=1 N1=1
20000000002f0000 default anon=7 dirty=7 N0=1 N1=6
<snip>
```

- Examine NUMA policy and page allocation for task <pid> address space
  - virtual address, policy [default], mapped file
  - anon/dirty page count
  - number of mapped pages, max mapcount [pte references]
  - NX=<pagecount> shows number of pages on node X for this virtual memory area

* per task numa_maps shows attributes of "vmas".
* "/proc/$$" == "my shell"
* Again, see /proc/<pid>/maps for additional vma info, such as range [start—end addresses], protections [RWX], usage [stack, heap, ...], path names of mapped files, ...

Any [more] Questions?